Package: matchr (via r-universe)

October 26, 2024

Type Package

Title Pattern Matching and Enumerated Types in R

Version 0.1.0

Author Christopher Mann <cmann3@unl.edu>

Maintainer Christopher Mann <cmann3@unl.edu>

Description Inspired by pattern matching and enum types in Rust and many functional programming languages, this package offers an updated version of the 'switch' function called 'Match' that accepts atomic values, functions, expressions, and enum variants. Conditions and return expressions are separated by '->' and multiple conditions can be associated with the same return expression using '|'. 'Match' also includes support for 'fallthrough'. The package also replicates the Result and Option enums from Rust.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 3.5.0), rlang, utils

LazyData true

RoxygenNote 7.1.1

NeedsCompilation no

Date/Publication 2021-09-09 09:10:02 UTC

Repository https://cmann3.r-universe.dev

RemoteUrl https://github.com/cran/matchr

RemoteRef HEAD

RemoteSha e28e696695b1f8fa0d76295d64704be50c74fdfe

Contents

---

bang *Extract Result or Return*

Description

Returns the value contained inside of an Result or Option Enum or returns if failure.

Usage

```
## S3 method for class 'Result'
!x, ...

## S3 method for class 'Option'
!x, ...
```

Arguments

x	Enumerated value of type Result or Option to unwrap
...	objects to be passed to methods.

Details

This is similar to `unwrap` for Result and Option objects. However, an `Err` or `None` variant does not cause execution to stop. Instead, the parent function immediately returns the Enum intact. Inspired by the ? operator in Rust.

Value

an object of any class or `x` if failure.

Functions

- `!.Result`: Unwrap Result if Ok, otherwise return the Err variant in the parent function.
- `!.Option`: Unwrap Option if Some, otherwise return the None variant in the parent function.

Examples

```
is_big <- function(x) {
  if (x > 10) return(Ok(x))
  Err("This is small!")
}

# If 'x' is greater than 10, the value will be printed.
# Otherwise, an error is returned.
print_big <- function(x) {
  print(!is_big(x))
}
```

`Enum` *Create Enumerated Type*

Description

An object inspired by enums in Rust and types in other functional languages.

Usage

```
Enum(...)
```

Arguments

`...`	Symbols specifying the named of the variant, or language call with the names and default values of objects contained within the variant. Other values can be used as long as the variant is named. The first item in `...` can optionally be a character string that names Enum.

Details

The Enum function creates a list of objects of class "Enum" *or* functions that generate "Enum" objects similar to those found in Rust of similar languages. Symbols or characters passed to Enum become the new variants. Language objects, i.e. a name followed by parentheses name(...), associate the name with the variant and create a function based on the arguments passed in .... When function is called, the passed arguments are converted into a named list of class "Enum" and associated variant. Like functions, default values can be given to the variants.

Variants can be assigned specific values using '='. For example, Enum(Hello = "world") creates an enum variant named "Hello" with the underlying value of "world". If the initial variant is assigned a single numeric value, then subsequent variants are automatically assigned the next highest value if possible, similar to using iota() in Go. Variant names are not allowed to be numeric values or other non-symbolic values.

Value

a list of variants or variant generators

Examples

```
### Create a Linked List

# Node is an enum with two varieties: a link to the next node, and none
# 'Node$Some' is a function that accepts two values and generates the enum
# variant, while 'Node$Empty' is a variant
Node <- Enum(
  Some(Val, Next),
  Empty
)

# Initialize an empty linked list, push values to the front
new_list <- Node$Empty
new_list <- Node$Some(1, new_list)
new_list <- Node$Some(2, new_list)
new_list

# return the head of the list ('car') and tail ('cdr')
car <- new_list$Val
cdr <- new_list$Next


### RGB Colors

# The Color enum is provided with a named type "Color". All
# variants will have both "Enum" and "Color" as a class.
# Each variant is associated with a specific value.
Color <- Enum(
  "Color",
  Black = c(0,0,0),
  Red   = c(255,0,0),
  Green = c(0, 255, 0),
  Blue  = c(0, 0, 255),
```

```
  White = c(255, 255, 255)
)

Color$Red

# This will generate an error since it is not a function
# Color$Black()


### Directions

# This enum creates a sequence of numbers associated with
# a particular direction. Enum automatically increments the
# values if the initial variant is assigned a single number
Direction <- Enum(
  North = 1,
  East,
  South,
  West
)

# This will result in '5' since North is '1' and West is '4'
Direction$North + Direction$West
```

`enum_type` *Enum Type*

Description

Return the enumerated type name of an object, if a name was provided.

Usage

```
enum_type(x, ...)
```

Arguments

`x`	Enum object
`...`	objects passed to methods

Value

character with the name of the enumerated type or `NULL`

Examples

```
x <- Result$Ok("hello world!")
enum_type(x)  # "Result"
```

Err *Create an 'Err' Result*

Description

Create an Enum variant of `Result` used to denote that function contained an error. This allows the creation of safer functions that do not automatically stop, without using `try` or `tryCatch`.

Usage

```
Err(e)
```

Arguments

`e`	Object to be wrapped in the Enum variant.

Value

a list with a single value `e` and classes `"Result}` and `\code{"Enum`

Examples

```
grepl_safe <- function(pattern, x)
{
 if (!is.character(pattern)){ return(Err("'pattern' in 'grepl_safe' was not a character value.")) }
  if (!is.character(x)){ return(Err("'x' in 'grepl_safe' was not a character value.")) }
  Ok(grepl(pattern, x))
}

#grepl_safe(123, 1:5)
```

fallthrough *Fall Through Match*

Description

Stop execution of current return expression in `Match`, then continue attempting to match conditions.

Usage

```
fallthrough()
```

Value

Object of class 'fallthrough'

Examples

```
Match(
  "abc",
  is.character -> {
    print("Found a character.")
    fallthrough()
  },
  "abc" -> "start of the alphabet",
  .     -> "found nothing"
)
```

`into_option` *Convert Object into Option*

Description

Create an `Option` out of an object. By default the object is wrapped in a `Some` variant. `Ok` variants of `Result` are turned into `Some` Options, while `Err` variants are turned into `None` Options.

Usage

```
into_option(x, ...)
```

Arguments

`x`	Object to be converted
`...`	Objects passed to methods

Value

an Enum object of class `Option`

Examples

```
an_error <- Result$Err("hello world!")
into_option(an_error) # None
```

into_result *Convert Object into Result*

Description

Create a `Result` out of an object. By default the object is wrapped in an `Ok` variant. `Some` variants of `Option` are turned into `Ok` Results, while `None` variants are turned into `Err` Results

Usage

```
into_result(x, ...)
```

Arguments

`x`	Object to be converted
`...`	Objects passed to methods

Value

an Enum object of class `Result`

Examples

```
nothing <- Option$None
into_result(nothing) # Err
```

is.enum *Is Object an Enum*

Description

Test whether object has class `Enum`.

Usage

```
is.enum(x)
```

Arguments

`x`	object to be tested

Value

`TRUE` if `x` is an Enum, `FALSE` otherwise

Examples

```
HelloEnum <- Enum(
  "HelloEnum",
  Hello,
  World
)

# TRUE
is.enum(HelloEnum$Hello)

# FALSE
is.enum(5)
```

`is.enum_type` *Check Enum Type*

Description

Test whether `Enum` is also of class `type`.

Usage

```
is.enum_type(x, type, ...)
```

Arguments

`x`	object to be tested
`type`	character string denoting type to check.
`...`	objects passed to methods

Value

`TRUE` if `x` has enumerated type `type`, `FALSE` otherwise

Examples

```
HelloEnum <- Enum(
  "HelloEnum",
  Hello,
  World
)

# TRUE
is.enum_type(HelloEnum$Hello, "HelloEnum")

# FALSE
is.enum_type(HelloEnum$Hello, "Hello")
```

is.err *Check if Result is an Err*

Description

Test whether Result Enum is Ok or an Err.

Usage

```
is.err(x)
```

Arguments

x	object to be tested

Value

TRUE if x is enumerated type of variant Err, FALSE otherwise

Examples

```
sqrt_big <- function(x) {
  if (x > 1000){ return(Ok(sqrt(x))) }
  Err("Not large enough!")
}
x <- sqrt_big(250)
is.err(x)  # TRUE
```

is.none *Check if Option is None*

Description

Test whether Option Enum is Some or None.

Usage

```
is.none(x)
```

Arguments

x	object to be tested

Value

TRUE if x is enumerated type of variant None, FALSE otherwise

Examples

```
x <- 1:5
get_n <- function(x, n) {
  if (n > length(x)) return(None)
  Some(x[n])
}
obj <- get_n(x, 6)
is.none(obj)  # TRUE
```

`is.ok` *Check if Result is Ok*

Description

Test whether Result Enum is Ok or an Err.

Usage

```
is.ok(x)
```

Arguments

`x`	object to be tested

Value

`TRUE` if `x` is enumerated type of variant `Ok`, `FALSE` otherwise

Examples

```
sqrt_big <- function(x) {
  if (x > 1000){ return(Ok(sqrt(x))) }
  Err("Not large enough!")
}
x <- sqrt_big(250)
is.ok(x)  # FALSE
```

is.some *Check if Option is Some*

Description

Test whether Option Enum is Some or None.

Usage

```
is.some(x)
```

Arguments

x	object to be tested

Value

TRUE if x is enumerated type of variant Some, FALSE otherwise

Examples

```
x <- 1:5
get_n <- function(x, n) {
  if (n > length(x)) return(None)
  Some(x[n])
}
obj <- get_n(x, 6)
is.some(obj)  # FALSE
```

is.variant *Check Enum Variant*

Description

Test whether Enum is variant variant.

Usage

```
is.variant(x, variant, ...)
```

Arguments

x	object to be tested
variant	character string denoting variant to check.
...	objects passed to methods

Value

TRUE if x is enumerated type of variant variant, FALSE otherwise

Examples

```
HelloEnum <- Enum(
  "HelloEnum",
  Hello,
  World
)

# TRUE
is.variant(HelloEnum$Hello, "Hello")

# FALSE
is.variant(HelloEnum$Hello, "World")
```

Match *Match Value Against Multiple Values*

Description

Functional programming style matching using -> to separate conditions from associated return values.

Usage

```
Match(x, ...)
```

Arguments

x	object to match
...	conditions used for matching, separated from the returned value by ->. Multiple conditions can be associated with the same return value using |. Each matching statement must be separated by a comma. See "Details" below. Use . to represent the default *(else ...)* condition.

Details

Unlike switch, Match accepts a variety of different condition statements. These can character, numeric, or logical values, functions, symbols, language objects, enums, etc. For example, "hello" -> 1 tests whether the object is equal to "hello". If so, the function returns 1, otherwise the next condition is tested. <- can also be used. If so, the condition & return expression are reversed: 1 <- "hello" also tests "hello" and returns 1.

Each condition is tested sequentially by calling the appropriate method of match_cond. If the condition is a character value, then match_cond.character is called, and so on. If a match is confirmed, the right-hand side is evaluated and returned.

For atomic vectors - numeric, logical, or character - `Match` will check for equality. All resulting values must be `TRUE` to match. Lists and environments are checked using `identical`. If a function is placed within the condition, then the function will be evaluated on object `x`. If the result is logical and `TRUE`, then it is considered a match. A non-logical result will be checked again using `match_cond`. Failed function calls with an error are treated as a non-match rather than stopping `Match`. Expressions are evaluated similar to functions.

The period `.` is a special condition in `Match`. When alone, it is treated as the "default" condition that always matches. When used as a call, though, it matches values within object `x` and/or attaches the individual items within `x` for use in the return expression. For example, `x = c(1, 2)` will be matched with the condition `.(1, second)`. This is because the first values are identical `(1 == 1)`. Furthermore, `second = 2` for use in the return expression. Preface a symbol with `..` to evaluate it and check for equality. `...` can be used to denote any number of unspecified objects.

The period call `.()` can also be used to test named member of `x`, though all objects in `.()` must be named to do so. For example, the condition `.(a = 5, b=)` tests whether `x` contains `"a"` with a value of `5` and `"b"` with any value.

If `function(...)` is used on the left hand side, then it may need to be surrounded by parentheses for the parser to properly recognize it. The `%fn%` infix function has be provided as syntactic sugar for developing functions for matching.

Similar to many functional languages, `(first:rest)` can be used as a condition to extract the first element and the rest from any vector as long as the vector is sufficiently long. Variables used on the left hand side can be called on the right hand side expression.

Matching an `Enum` causes symbols to represent possible variants. For example, `None -> "none"` would try to match the variant of `x` with `None`. If it succeeds, then `Match` will return `"none"`. A function call on the left-hand side for an Enum is treated as a variant and its inside arguments, which are made available in the result expression. So, `Some(var) -> sqrt(var)` would attempt to match on the variant `Some`. If it matches, then the inside is exposed as the variable `var` for the right-hand side to use. The number of objects in the variant on the left-hand side must match the number of objects inside of `x` or else an error will populate.

Regex conditions can be used when matching strings by surrounding the expression in braces. For example, the condition "[ab]*" is equivalent to using `grepl("\[ab\]*", ...)`. The braces must be the first and last characters to trigger a regex match.

Call `fallthrough` within a return expression to stop evaluating the expression and return to matching. This can be convenient for complex matching conditions or to execute code for side-effects, such as printing.

Value

an object based on the matched clause. An Error is produced if no match is found.

Examples

```
## Matching to functions, characters, regex, and default
Match(
 "abc",
 is.numeric       -> "Not a character!",
 is.character     -> {
   print("Found a character!")
```

```
   fallthrough()
 },
 "a" | "b" | "c"  -> "It's a letter!",
 "{bc}"           -> "Contains 'bc'!",
 .                -> "Can be anything!"
)


## Unwrapping a Result enum
val <- Result$Ok("hello world!")

Match(
  val,
  Ok(w)  -> w,
  Err(s) -> s
)

## Using functions
# If 'function' is used on the lhs, surround in '()'
# Alternatively, use %fn% notation
Match(
  1:10,
  (function(i) mean(i) < 5)  -> TRUE,
  i %fn% (mean(i) >= 5) -> FALSE
)

## Extracting parts
x <- list(a = 5, b = 6, c = 7)
Match(
  x,
  .(a=, d=2)  -> "won't match, no 'd'",
  .(a=5, b=)  -> "will match, a == '5'",
  (x:xs)      -> {
    print(x)  # 5
    print(xs) # list(b=6, c=7)
    "will match, since not empty"
  },
  .           -> "this matches anything!"
)

z     <- c(1,2,3,4)
first <- 1
Match(
  z,
  .(0, ...) -> "no match, first is 1 not 0",
  .(1, 2)   -> "no match, z has 4 elements",
  .(x, 2, ...) -> paste("match, x = ", x),
  .(..first, ...) -> "match, since 'first' == 1"
)
```

Matchply *Match Each Object in List or Vector*

Description

Applies Match to each individual object within the input rather than matching the entire object.

Usage

```
Matchply(x, ...)
```

Arguments

x	a vector (including list) or expression object
...	conditions and expressions for matching. See Match for details.

Details

See Match for details on condition implementation. Default conditions using the period . are highly recommended to prevent error.

Matchply is a wrapper to lapply and sapply, depending on the input object, with ... converted to a match statement for easy use.

Value

vector depending on input x. By default, sapply is used with simplify = TRUE. This could return a vector, matrix, list, etc. When simplify = FALSE or a list is provided, the result will be a list.

Examples

```
new_list <- list(
  hello = "World!",
  nice  = 2,
  meet  = "u"
)

Matchply(
  new_list,
  is.numeric  -> "found a number!",
  "{rld}"     -> "maybe found 'World'!",
  "u" | "z"   -> "found a letter",
  .  -> "found nothing"
)
```

match_cond *Check and Evaluate Match Condition*

Description

Called by `Match` the check whether a condition matches. Used to create custom methods for matching.

Usage

```
match_cond(cond, x, do, ...)
```

Arguments

`cond`	match condition
`x`	object being matched
`do`	return expression associated with the condition. If `cond` is matched with `x`, then `do` should be evaluated and returned in a list with `TRUE`: `list(TRUE, eval(do))`.
`...`	arguments passed to evaluation

Details

See the `Match` details for explanations about provided methods.

Value

`FALSE` if no match, or a list containing `TRUE` and the evaluated expression

None *None*

Description

An Enum variant of `Option` used to denote that a function returned no value.

Usage

```
None
```

Format

an empty list of classes `"Option"` and `"Enum"`

Ok *Create an 'Ok' Result*

Description

Create an Enum variant of `Result` used to denote that function did not contain an error. This allows the creation of safer functions.

Usage

```
Ok(x)
```

Arguments

x	Object to be wrapped in the Enum variant.

Value

a list with a single value x and classes "Result} and \code{"Enum

Examples

```
grepl_safe <- function(pattern, x)
{
 if (!is.character(pattern)){ return(Err("'pattern' in 'grepl_safe' was not a character value.")) }
  if (!is.character(x)){ return(Err("'x' in 'grepl_safe' was not a character value.")) }
  Ok(grepl(pattern, x))
}

#grepl_safe(123, 1:5)
```

Option *Option*

Description

An Enum that mimics Rust's "Option" type. This is used to denote whether a function returned an object or not, rather than returning NULL.

Usage

```
Option
```

Format

list with 1 Enum generators and 1 Enum variant

Some(x) Wrap x in the 'Some' variant.

None Variant denoting that nothing was returned.

Result *Result*

Description

An Enum that mimics Rust's "Result" type. This is used to denote whether a function contained an error without stopping execution and allowing the error result to be unwrapped.

Usage

```
Result
```

Format

list with 2 Enum generators

Ok(x) Wrap x in the 'Ok' variant.

Err(e) Wrap x in the 'Err' variant.

Some *Create an 'Some' Option*

Description

Create an Enum variant of `Option` used to denote that function returned a value. This allows the creation of safer functions that extract values from other objects, without using `try` or `tryCatch`.

Usage

```
Some(x)
```

Arguments

x	Object to be wrapped in the Enum variant.

Value

a list with a single value x and classes "Option} and \code{"Enum

Examples

```
subset_safe <- function(x, index) {
  if (index > length(x)){ return(None) }
  Some(x[index])
}
```

`Try` *Execute Expression as Result*

Description

Evaluates given expression returning an `Err` Result if there is an error, otherwise an `Ok` Result.

Usage

```
Try(expr)
```

Arguments

`expr`	expression to evaluate

Value

Result Enum of variant `Ok` or `Err`

Examples

```
# This returns an Err
Try(sqrt + 1)

# This returns an Ok
Try(sqrt(5) + 1)
```

`unwrap` *Extract the Value Contained in Enum*

Description

Returns the value contained inside of an enum variant. The function strips all relevant attributes from the object, returning its bare value.

Usage

```
unwrap(x, ...)

unwrap_or(x, alt, ...)
```

Arguments

x	Enumerated value to unwrap
...	objects to be passed to methods.
alt	Alternative value to be returned in case of failure

Details

unwrap is used to extract the inside objects of an Enum. Unless the Enum was assigned a specific value, the returned value will be a list with names equal to those in the Enum declaration.

Result and Option have associated unwrap methods that automatically call an error and stop execution if the variant is either Err(e) or None, respectively. unwrap_or allows the user to specify an alternative value in case of failure on the part of Result or Option.

Value

an object of any class.

Functions

- unwrap_or: Extract the inside of Enum. If variant is 'Err' or 'None', the alternative is returned.

Examples

```
Color <- Enum(
  "Color",
  Black = c(0,0,0),
  Red   = c(255,0,0),
  Green = c(0, 255, 0),
  Blue  = c(0, 0, 255),
  White = c(255, 255, 255)
)

red_rgb <- unwrap(Color$Red)
blue    <- rev(red_rgb)
blue

new_err <- Err("hello world!")
unwrap_or(new_err, "this is not an error")
```

---

variant *Enum Variant*

Description

Return the variant name of an enumerated type.

Usage

```
variant(x, ...)
```

Arguments

x	Enum object
...	objects passed to methods

Value

character with the name of the variant or NULL

Examples

```
x <- Result$Ok("hello world!")
variant(x)  # "Ok"
```

%.% *Compose Functions*

Description

Combine two functions into a single function so that the rhs is called on the arguments first, then the lhs.

Usage

```
lhs %.% rhs
```

Arguments

lhs	function to be called second
rhs	function to be called first

Value

a composed function

Examples

```
sq_log <- round %.% sqrt %.% log

Match(
  10:20,
  i %fn% (sq_log(i) > 2) ->
    "big",
  . ->
    "small"
)
```

`%fn%` *Create Function*

Description

Syntactic sugar for creating a single-variable function. Can be conveniently used in `Match` statements.

Usage

```
lhs %fn% rhs
```

Arguments

`lhs`	symbol used to denote the function argument
`rhs`	expression that is converted to the function body. `rhs` may need to be surrounded by parentheses is other infix operators are used due to precedence rules.

Value

a function

Examples

```
Match(
  "abc",
  is.numeric  -> -1,
  i %fn% grepl("bc", i) -> 0,
  is.character -> 1
)

print_sq_log <- i %fn% print(sqrt(log(i)))
print_sq_log(10)
```

Index